

AD-A242 539



2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
ELECTE
NOV 18 1991
S D D



THESIS

DATA COMPRESSION USING THE DICTIONARY
APPROACH ALGORITHM

by

Michael T. Kaoutskis

December, 1990

Thesis Advisor:

Chyan Yang

Approved for public release; distribution is unlimited.

91-15184



91 1106 028

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No 0704-0188	
1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited		
2b DECLASSIFICATION / DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b OFFICE SYMBOL (If applicable) EC		7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000			7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		
8a NAME OF FUNDING / SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)		9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
			WORK UNIT ACCESSION NO		
11 TITLE (Include Security Classification) DATA COMPRESSION USING THE DICTIONARY APPROACH ALGORITHM					
12 PERSONAL AUTHOR(S) KAOUTSKIS, Michael T.					
13a TYPE OF REPORT Master's Thesis		13b TIME COVERED FROM _____ TO _____		14 DATE OF REPORT (Year, Month, Day) 1990 December	
15 PAGE COUNT 84					
16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	entropy coding; compression ratio; decompression;		
			dictionary conversion into numbers		
19 ABSTRACT (Continue on reverse if necessary and identify by block number)					
Several data compression schemes have been investigated for reducing storage space and transfer time via a computer network.					
The primary goal of this thesis is to develop a new scheme for data compression ratio better than the already existing schemes. The main approach adopted by this research is a combination of dictionary look up and entropy source coding.					
20 DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL YANG, Chyan			22b TELEPHONE (Include Area Code) 408-646-2266		22c OFFICE SYMBOL EC/Ya

Approved for public release; distribution is unlimited.

Data Compression Using The Dictionary Approach Algorithm

by

Michael T. Kaoutskis
Lieutenant, Hellenic Navy
B.S., E.E, Hellenic Naval Academy 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

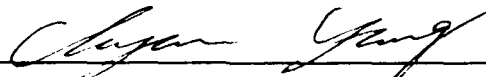
December 1990

Author:



Michael T. Kaoutskis

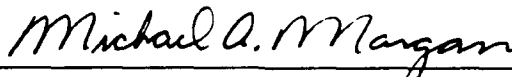
Approved by:



Chyan Yang, Thesis Advisor



Mitchell L. Cotton, Second Reader



Michael A. Morgan, Chairman

Department of Electrical and Computer Engineering

ABSTRACT

Several data compression schemes have been investigated for reducing storage space and transfer time via a computer network.

The primary goal of this thesis is to develop a new scheme for data compression with compression ratio better than the already existing schemes. The main approach adopted by this research is a combination of dictionary look up and entropy source coding.

Approved For	
J	
THIS PROGRAM	
DISTRIBUTION	
UNCLASSIFIED	
Justification	
By	
Distribution	
Approved For	
Date	
A-1	

TABLE OF CONTENTS

I. INTRODUCTION	1
II. ALGORITHMS FOR DATA COMPRESSION	3
A. THE FINITENESS OF SYMBOLS	3
B. RELATIVE FREQUENCY METHODS	4
1. Huffman Coding Algorithm	4
2. Arithmetic coding	6
C. RUN LENGTH ENCODING	7
D. PROGRAMMED COMPRESSION	8
E. ADAPTIVE CODE	9
1. Locally Adaptive Data Compression Scheme	9
2. L Z W Compression Algorithm.	12
a. Compression.	13
b. Decompression.	16
3. The LZ77 OPM/L Text Compression Technique.	17
F. DICTIONARY APPROACH	22
III. DICTIONARY DATA COMPRESSION.	23
A. OVERVIEW	23
B. COMPRESSION PROGRAM	24
1. Input Part.	24

2. Conversion Part	25
3. Numerical Translation	27
C. DICTIONARY	29
D. DECOMPRESSION PROGRAM	30
E. USAGE OF THE PROGRAM.	31
1. Compression	31
2. Decompression.	31
IV. STATISTICAL RESULTS - PERFORMANCE	33
A. GENERAL	33
1. Size of file vs. compression ratio	33
2. Comparison between different compression schemes	33
B. SIZE OF FILE VS. COMPRESSION RATIO	33
C. COMPARISON BETWEEN DIFFERENT COMPRESSION SCHEMES	36
D. NON-REVERSIBLE VERSION OF THE DICTIONARY PROGRAM	37
E. TIME IMPROVEMENT ALGORITHM	40
V. CONCLUSIONS	42
A. DATA COMPRESSION	42
B. FUTURE RESEARCH	42
APPENDIX A: PROGRAM LISTINGS	43
A. COMPRESSION PROGRAM	43
B. DECOMPRESSION PROGRAM	54
C. TIME IMPROVEMENT ALGORITHM	62

REFERENCES 72

INITIAL DISTRIBUTION LIST 73

LIST OF FIGURES

Figure 1 Huffman Coding	5
Figure 2 Coding CAT using Arithmetic Encoding.	7
Figure 3 The "move-to-front" word list.	11
Figure 4 The string and the alternate tables	14
Figure 5 The compression procedure	15
Figure 6 The decompression procedure	15
Figure 7 Performance of different compression schemes .	20
Figure 8 Binary searching tree for longest match . . .	21
Figure 9 Link list scheme for the Dictionary and Text	25
Figure 10 Conversion of words into numbers	27
Figure 11 Bit Conversion of Digits	27
Figure 12 Conversion of numbers and compose of character	28
Figure 13 Compression ratio vs. size of file with dictionary with less than 1000 words	34
Figure 14 Compression ratio vs. size file with dictionary with more than 3000 words	35
Figure 15 Comparison between the three schemes	36
Figure 16 Compression ratio vs. size of file with dictionary with less of 1000 words for non-reversible algorithm	37

Figure 17 Compression ratio vs. size of file for dictionary with more than 3000 words for non- reversible algorithm	38
Figure 18 Comparison between the non-reversible algorithm and the others compression schemes	39

ACKNOWLEDGMENT

I would like to express my gratitude and appreciation to the faculty and staff of the Electrical and Computer Engineering department for providing me with the opportunity and encouragement to explore many exciting facets of electrical engineering. I would like to offer special thanks to Professor Chyan Yang for providing the necessary guidance and direction in the formulation of this document. I also wish to thank Professor Mitchell Cotton and the lab technicians Elaine Codres, Bob Limes, and Gary Rediske for their assistance.

Finally, I am most grateful to my wife Elina for her patience, understanding, and support during my studies.

I. INTRODUCTION

This thesis presents an initial step in developing a new data compression technique based in the dictionary approach. "Data compression is the process of encoding a body of data D into a smaller body $\Omega(D)$." [Ref. 7]. If $\Omega(D)$ can be decoded back to D , without loss of information, then it is said to be a reversible data compression. The situation in which some acceptable approximation to D is obtained in the decoding is known as non-reversible data compression.

Usually non-reversible algorithms are used for image compression. This thesis considers only ASCII text compression techniques.

The primary advantages to utilizing data compression techniques are:

- . Data storage space such as disks or tapes can be greatly reduced with data compression. A compressed file generally takes less storage space than an uncompressed one. Also, compressed files can be decompressed when users demand the original copy.
- . With data compression the same amount of information can be sent over a network in much less time than decompressed data. For data

communications, a sender can compress data before transmitting it and the receiver can decompress the data after receiving it.

The main parameters of interest in data compression are compression ratio $C(r)$ and compression speed. Compression ratio is defined as

$$\frac{\text{amount of compressed data}}{\text{amount of original data}} \times 100$$

Therefore, a compression ratio of 33% would mean that the compressed text is one-third the size of the original text. The compression speed is the average bytes compressed per second. These two performance measures are often inversely proportional to each other. The trade off between them depends on the application requirements.

II. ALGORITHMS FOR DATA COMPRESSION

Data compression can be approached in three ways:

- . The finiteness of symbols.
- . The relative frequencies with which the symbols are used.
- . The context in which a symbol appears.

This chapter will examine in detail, each of these three approaches to data compression.

A. THE FINITENESS OF SYMBOLS

One example of a finite set are the titles of the books that exist in a library. Usually a title of book has about 20 characters. If the title is translated via the ASCII code, 140 bits (one character needs 7 bits) are required. However if each title is given a sequence number, then the sender can send the sequence only and the receiver can map this number into the appropriate title. The largest library in the world has about 2^{30} titles (The Library of Congress has 2^{26} titles)[Ref. 9]. Using sequence numbers instead of titles, the necessary bits to transmit are only 30 instead of 140. With this method we have a compression ratio $(30/140)*100 = 21\%$. Here we assume that the sender and the receiver have the same translation table that translate the titles into numbers.

B. RELATIVE FREQUENCY METHODS

An important parameter of data transmission is the **entropy** of a symbol S of an alphabet $\Sigma\{S_1, S_2, S_3, \dots, S_k\}$ given by the formula: [Ref. 4]

$$H_r(s) = \Sigma p_i \log(1/p_i) \quad (1)$$

where p_1, p_2, \dots, p_k are the probabilities of occurrence of each symbol that contained in alphabet Σ . When the radix r is not given we assume $r=2$ and abbreviate $H_2(S) = H(S)$.

The importance of entropy is that for a source coding there exists an inherent entropy that cannot be exceeded. The notion of entropy provides a foundation for intuitively reasonable facts such as:

- . Random data cannot be compressed.
- . Data that has been compressed by an optimal compressor (one that always achieves the entropy of the source) cannot be compressed further.
- . One cannot guarantee that a data compressor will achieve any given performance on all data.

1. Huffman Coding Algorithm

The Huffman code uses the frequency of occurrence that a symbol appears in the text. The most frequently used symbols have a shorter binary pattern and less frequently symbols have a longer pattern. For example, suppose we want to compress the text:

" T h i s i s a t e s t t e x t "

The set of symbols and their relative frequencies are:

$P(T) = 5/15$ $P(h) = 1/15$ $P(i) = 2/15$ $P(s) = 3/15$

$P(a) = 1/15$ $P(e) = 2/15$ $P(x) = 1/15$.

T--> 5 00	5 00	5 00	5 00	6 1	9 0
s--> 3 11	3 11	3 10	4 01	5 00	6 1
i--> 2 011	2 010	3 11	3 10	4 01	
e--> 2 100	2 011	2 010	3 11		
h--> 1 101	2 100	2 011			
x--> 1 0100	1 101				
a--> 1 0101					

Figure 1 Huffman Coding

Each letter can be coded with the following bit pattern show in Fig. 1, $T = 00$, $s = 11$, $i = 011$, $e = 100$, $h = 101$, $x = 0100$, and $a = 0101$. Note that the most frequent symbols are translated with only two bits and less frequent ones, such as x , with four.

One disadvantage of the Huffman code and all codes that based on the frequency of existence is that the source text must be scanned twice before transmission begins. The first

pass determines the frequencies of occurrence of each symbol and the second compresses the text.

2. Arithmetic coding

Arithmetic coding is based on the idea that each symbol is not coded independently one after another as in Huffman code, but coded as a portion of the real number line between 0 and 1. Encoding a sequence of symbols ultimately results in selecting a portion of the reals and transmitting a number in that portion.

The operation of the algorithm can most easily be seen with an example. Four symbols A, C, G, and T, occur with frequencies 0.50, 0.30, 0.15, and 0.05, respectively. The sum of the probabilities is 1. With this probability distribution, the interval 0.00 to 0.50 is used for A, 0.50 to 0.80 for C, 0.80 to 0.95 for G, and 0.95 to 1.00 for T. Suppose that we want to encode the symbol string CAT as in Fig.2. For the first symbol C, the range is narrowed to the interval 0.50 to 0.80. As the algorithm progresses the interval is steadily narrowed, requiring more and more bits provide the necessary accuracy. The next symbol to encode is A, which utilizes the interval 0.50 to 0.65 of Fig. 2(b). If we expand this to unit length as in Fig. 2(c) and select the T part from it, we get the interval 0.6425 to 0.6500. If there were more symbols, they would further subdivide the interval again as shown in

Fig. 2(d). The encoded result that is transmitted is any value in the final range, for example 0.645.

When the receiver gets this value, it immediately sees that

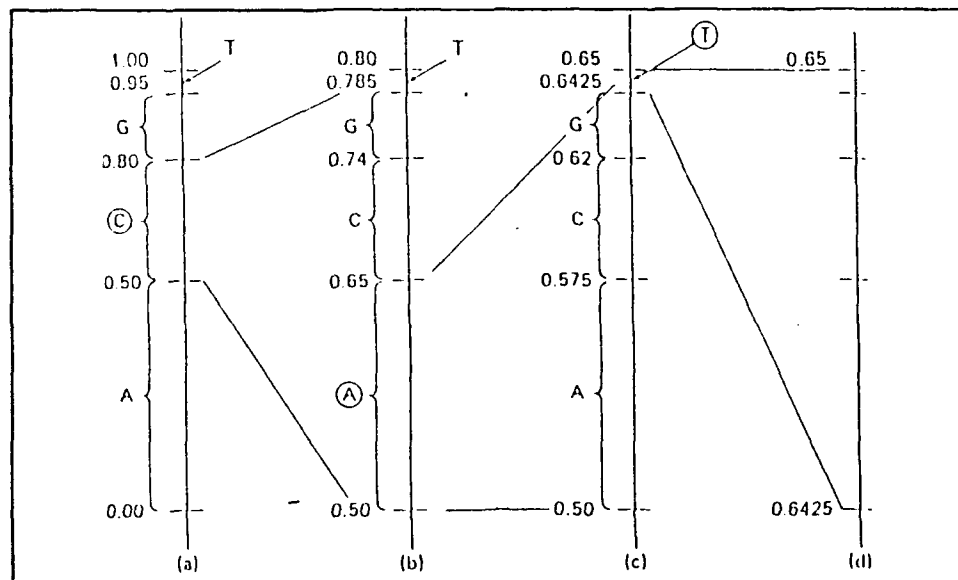


Figure 2 Coding CAT using Arithmetic Encoding.

it lies between 0.50 and 0.80, indicating that the first symbol must be a C. It then constructs the interval 0.50 to 0.80, just as the sender did, and sees that 0.645 lies between 0.50 and 0.65, meaning that the second symbol must be an A. In this manner, the receiver decodes the message, symbol by symbol.

C. RUN LENGTH ENCODING

This technique based on the probability of existence of a symbol if the previous one is known. This method is primarily used to encode long binary bit strings containing mostly

zeros. Each K-bit symbol tells how many zero bits occurred between consecutive 1 bits. To handle long zero runs, two symbols consisting of all 1 bits means that the true distance is $2^k - 1$ plus the value of the following symbol. Consider the following binary bit pattern:

000100000100000010000000000000010000001000100000001101000001

which consists of zero runs of length 3,5,6,14,6,3,7,0,1, and 5. It can be encoded using 3-bit symbol as:

011	101	110	111	111	000	110	011	111	000	000	001	101
1	1	1				1	1	1		1	1	1

The bit pattern 111 000 means that there are 7 zeros followed by one 1. The bit pattern 000 indicates two consecutive ones. Using this compression scheme it can be shown that a reduction of approximately 32% is possible.

D. PROGRAMMED COMPRESSION

Programming is generally done by the applications programmer or data management system. In formatted data files, several techniques are used. Unused blank or zero spaces are eliminated by making fields variable in length and using an index structure with pointers to each field position. Predictable field values are compactly encoded by way of a code table - such as when warehouse names are given as integer

codes rather than an alphabetic English names. Each field has its own specialized code table that deals with positional redundancy. Since programmed compression cannot effectively handle character distribution redundancy, it is a nice complement to Huffman encoding.

Programmed compression has several serious disadvantages. It introduces increased program development expenses; the type of decompression used requires a knowledge of the record structure and the code tables; the choice of field sizes and code tables may complicate or inhibit later changes to the data structure making the software more expensive to maintain.

E. ADAPTIVE CODE

The adaptive data compression algorithm is a scheme that exploits locality of reference: words are used frequently over short intervals and then fall into long periods of disuse. There are many different algorithms like **Lempel-Ziv** [Ref. 13] technique or **L Z W** [Ref. 11] technique or a **Locally Adaptive Data Compression Scheme** [Ref. 2] by BENTLEY SLEATOR TARJAN and WEI, and many others.

1. Locally Adaptive Data Compression Scheme

This technique is based on a simple heuristic for a self organizing sequential search and on variable-length encodings of integers. This scheme has the advantages that it is simple, allows fast encoding and decoding, and requires

only one pass over the data to be compressed. (Huffman takes two passes).

As mentioned earlier, this scheme is based on the locality of reference meaning that in a certain text some words appear with high frequency at one point of the text and with low frequency in another point. Therefore this algorithm is based on a self-organizing search which maintains a sequential list of words with frequently accessed words near the front. This data compression scheme uses a self-organizing list as an auxiliary data structure and employs short encoding to transmit words near the front of the list and long encoding for the words at the end of the list.

Example:

Suppose we want to compress the message:

THE CAR ON THE LEFT HIT THE CAR I LEFT 0

(The words are written with capital letters, separates with a single space, and the end of the message is indicated by the symbol 0).

The sender and receiver maintain identical word lists using the " move-to-front " heuristic: After a word is used it is deleted from its current position and moved to the front of the list. This attempts to ensure that frequently used words appear near the front of the list.

THE CAR ON	THE	LEFT	HIT	THE	CAR	I	LEFT
1. ON	1.THE	1.LEFT	1.HIT	1.THE	1.CAR	1.I	1.LEFT
2.CAR	2.ON	2.THE	2.LEFT	2.HIT	2.THE	2.CAR	2.I
3.THE	3.CAR	3.ON	3.THE	3.LEFT	3.HIT	3.THE	3.CAR
		4.CAR	4.ON	4.ON	4.LEFT	4.HIT	4.THE
			5.CAR	5.CAR	5.ON	5.LEFT	5.HIT
						6.ON	6.ON

Figure 3. The "move - to - front" word list.

After the construction of the word list as in Fig. 3, the sender transmit the following message:

1 THE | 2 CAR | 3 ON | 3 | 4 LEFT | 5 HIT | 3 | 5 | 6 I | 5 |.

The list is initially empty. To transmit the word W, the sender looks it up in the list. If it is present in position L, the sender transmits L, which the receiver decodes by writing the Lth element in the list; both then move W to the front of their respective lists, shifting the words in positions 1,2...L-1 to positions 2,3..L. If W is not in the list of N words, the sender reacts as though it were in the N+1st position and sends the integer N+1 followed by the word W (which the receiver expects because N+1 is greater than the size of the current list); both sender and receiver then move

W to the front of their lists. Each word is transmitted as a string of letters just once; subsequent occurrences are encoded by integers. The integer encoding of a word is one greater than the total number of different words that have occurred since its previous appearance.

The above example illustrates the most important property of the scheme: if a word has been recently used then it will be near the front of the list and therefore have a short decimal encoding. Because the integer L requires roughly $\log_{10} L$ characters to encode, frequent words are transmitted with few characters. This scheme has many variations, one of them is the L Z W technique.

2. L Z W Compression Algorithm

The L Z W algorithm is organized around a translation table, referred to as a string table (instead of a word list as in the previous scheme), that maps strings of input characters into **fixed-length** codes. The use of 12-bit codes is common. The L Z W string table has a prefix property in that for every string in the table its prefix string is also in the table. That is if string δM , composed of some string δ and some single character M , is in the table, then δ is in the table. M is called the **extension character** on the prefix string δ . The string table in this explanation is initialized to contain all single - character strings.

The L Z W string table contains strings that have been encountered previously in the message being compressed. It consists of a running sample of strings in the message, so the available strings reflect the statistics of the message.

L Z W uses the "greedy" parsing algorithm, where the input string is examined character-serially in one pass, and the longest recognized input string is parsed off each time. A recognized string is one that exists in the string table. Strings added to the string table are determined by this parsing: Each parsed input string extended by its next input character forms a new string added to the string table. Each such added string is assigned a unique identifier, namely its code value.

a. Compression.

The compression algorithm in each execution parsed off an acceptable string δ . The next character M is read and the extended string δM is tested to see if it exists in the string table. If it is there, then the extended string becomes the parsed string δ and the step is repeated. If δM is not in the string table, then it is entered, the code for the successfully parsed string δ is put out as compressed data, the character M becomes the beginning of the next string, and the step is repeated. An example of this procedure is shown in Figures 4, 5, 6. For simplicity a three-character alphabet is used.

STRING TABLE		ALTERNATE TABLE	
a	1	a	1
b	2	b	2
c	3	c	3
-----		-----	
ab	4	1b	4
ba	5	2a	5
abc	6	4c	6
cb	7	3b	7
bab	8	5b	8
baba	9	8a	9
aa	10	1a	10
aaa	11	10a	11
aaaa	12	11a	12

Figure 4 The string and the alternate tables

The string table is initialized with three code values for the three characters, shown above the dotted line. Code values are assigned in sequence to new strings. The alternate table is constructed from the code value of the existing string and the new character M that was added.

The compression procedure is shown in fig. 5. The input data, being read from left to right, is examined starting with the first character a. Since no matching string longer than a exists in the table, the code 1 is output for this string and the extended string ab is put in the table under code 4. Then b is used to start next string. Since its extension ba is not in the table, it put there under code 5, the code for b is

output, and **a** starts the next string. This process continues straightforwardly.

INPUT SYMBOLS	a	b	ab	c	ba	bab	a	aa	aaa
OUTPUT CODES	1	2	4	3	5	8	1	10	11
NEW STRING ADDED TO TABLE			5	7		9		11	
	4		6		8		10		

Figure 5 The compression procedure

For the decompression each code is translated by recursive replacement of the code with the prefix code and extension character from the string table (Fig. 4). For example code 5 is replaced by code 2 and **a**, and then code 2 is replaced by **b**.

INPUT CODES	1	2	4	3	5	8	1	10	11
	v	v	v	v	v	v	v	v	v
	a	b	1b	c	2a	5b	a	1a	10a
			v		v	v		v	v
			a		b	2a		a	1a
						v			v
						b			a
OUTPUT DATA	a	b	ab	c	ba	bab	a	aa	aaa
STRING ADDED TO TABLE	4		6		8		10		
		5		7		9		11	

Figure 6 The decompression procedure

This algorithm makes no real attempt to optimally select strings for the string table or optimally parse the input data. It produces compression results that, while less than optimum, are effective. Since the algorithm is clearly quite simple, its implementation can be very fast.

The principal concern in implementation is storing the string table. To make it tractable, each string is represented by its prefix string identifier and extension character, so each table entry has **fixed length**.

b. Decompression.

The L Z W decompressor logically uses the same string table as the compressor and similarly constructs it as the message is translated. Each received code value is translated by way of the string table into a prefix string and extension character. The extension character is pulled off and the prefix string is decomposed into its prefix and extension. This operation is recursive until the prefix string is a single character, which completes decompression of that code Fig. 6. This terminal character, called the final character, is the left-most character encountered by the compressor when the string was parsed out.

An update to the string table is made for each code received (except the first one). When a code has been translated, its final character is used as the extension character, combined with the prior string, to add a new string

to the string table. This new string is assigned a unique code value, which is the same code that the compressor assigned to that string. In this way, the decompressor incrementally reconstructs the same string table that the compressor used.

3. The LZ77 OPM/L Text Compression Technique

The LZ77 is an OPM/L data compression scheme suggested by Ziv and Lempel. A slightly modified version of this scheme improving the compression ratios for wide range of texts is developed by Storer and Szymanski and called **LZSS** with very fast decoding and comparatively little memory required for coding and decoding.

An OPM/L (original pointer macro restricted to left pointers) scheme replaces a substring in a text with a **pointer** to a previous (**left**) occurrence of the substring in the text. The pointer represents the position and size of the sub-string in the **original** text. These restrictions make fast single-pass decoding straightforward.

The LZ77 scheme restricts the reach of the pointer to approximately the previous N characters, effectively creating a "**window**" of N characters which are used as a sliding dictionary. Pointers are chosen using a "**greedy**" algorithm which permits single-pass encoding.

The use of a window has several advantages:

- . The amount of memory required for encoding and decoding is bound by the size of the window, and is typically no more than 8 kbytes.
- . For many types of text, and for sufficiently large N , the window is a good dictionary for the substring which follows because it will usually contains the same language, style and topic.
- . All pointers can have fixed size fields.

An LZ77 encoder is parameterized by N , the size of the "window", and F , the maximum length of a substring that may be replaced by a pointer. Encoding of the input string proceeds from left to right. At each step of the encoding a section of the input text is available in a window of N characters. Of these, the first $N-F$ characters have already been encoded and the last F characters are the "lookahead buffer".

For example, if the string $s = \text{abcabcbacbababcabc...}$ is being encoded with the parameters $N = 11$ and $F = 4$ and character 12 is to be encoded next, the window is:

5	6	7	8	9	10	11	12	13	14	15
b	c	b	a	c	b	a	b	a	b	c
already encoded						lookahead buffer				

Initially the first $N-F$ characters of the window are (arbitrarily) blanks, and the first F characters of the text are loaded into the lookahead buffer.

The already encoded part of the window is searched to find the longest match for the lookahead buffer, but obviously

cannot be the lookahead buffer itself. In the example, the longest match for the "bab" is "bab", which starts at character 10.

The longest match is then coded into a triple $\langle i, j, a \rangle$, where i is the offset of the longest match from the lookahead buffer, j is the length of the match, and a is the first character which did not match the substring in the window. In the example, the output triple would be $\langle 2, 3, 'c' \rangle$. The window is then shifted right $j + 1$ characters, ready for another coding step.

Decoding is very simple and fast. The decoder maintains a window in the same way as the encoder but, instead of searching for a match in the window uses the triple given by the encoder.

The main disadvantage of LZ77 is that, although the encoding step requires $O(1)$ time, a straightforward implementation can require up to $(N-F)*F$ character comparisons, typically on the order of several thousands. LZ77 is therefore best for the situation where a file is to be encoded once (preferably on a fast computer) and decoded many times, possibly on a small machine. Examples of these situations are on-line help files and manuals, decentralized databases, teletext, and electronic books.

Figure 7 lists the performance of the different compression schemes with the parameters of speed and memory they use for the compression.

		LZSS N=8192	LZSS N=2048	LZ77 N=8192	LZ78	LZW	Arith- meic	Adapt. Huff.
Speed (characters per second)	Encode	18	52	24	5300	5700	-	990
	Decode	13600	10900	15200	10060	8400	-	1300
Memory (kbytes)	Encode	8	2	8	350	48	³² 1400	8
	Decode	8	2	8	135	12	³² 1400	8

Figure 7 Performance of different compression schemes

One improved technique for reducing the time for compression is suggested by T.C.BELL [Ref. 1] since time is the only point that LZ77 or LZSS techniques fall short of the other algorithms that shown in Fig. 7. The algorithm developed by BELL is the "binary tree algorithm" that searches for the longest match for a string.

Consider the same string as in LZ77 technique:

$S = a b c a b c b a c b a b a b c a b c \dots$ with parameters $N = 11$ and $F = 4$.

5	6	7	8	9	10	11	12	13	14	15
b	c	b	a	c	b	a	b	a	b	c

The lookahead buffer is defined as $l = x_1 = babc$ and $x_5 = bcba$ $x_6 = cbac$ $x_7 = bacb$ $x_8 = acba$ $x_9 = cbab$ $x_{10} = baba$ $x_0 = abab$. By inspection the longest match is x_{10} with vector $(1, x) = (10, 3)$ where 10 is the position of the match string start and 3 is the characters that match the lookahead buffer.

The binary search algorithm start with sorting the x_5 , x_6 , ... x_0 with lexicographical order. So we have:

x_0	x_8	x_{10}	1	x_7	x_5	x_9	x_6
abab	acba	baba	babc	bacb	bcba	cbab	cbac

The longest match for 1 should be found at the beginning of x_{10} or x_7 . This happened because these two strings are lexicographically adjacent to the lookahead buffer 1 and are the two candidates for the longest match.

The basic construction of the tree is that for any node x_i all nodes in its left subtree are lexicographically less than x_i and all nodes in its right subtree are lexicographically greater than x_i . So with this way the tree is constructed starting with x_5 , x_6 , x_7 , ... x_{10} , x_0 , 1 and then x_{10} , and x_7 appear on the path to 1 as shown in Fig. 8.

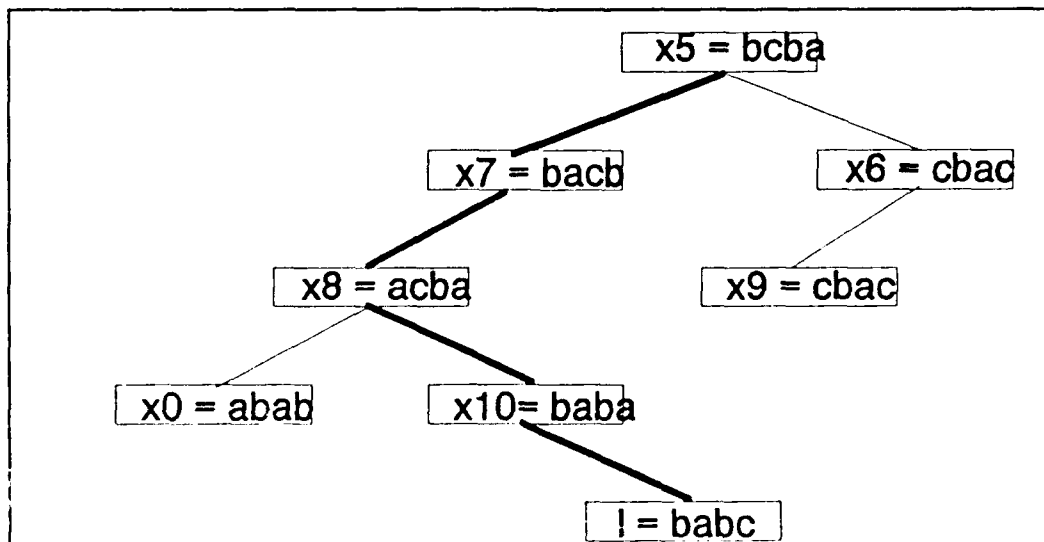


Figure 8 Binary searching tree for longest match

F. DICTIONARY APPROACH

This approach is based on the fact that the sender and the receiver have the same dictionary of words, each word can be encoded by its coding number. Each coded number can be uniquely decoded. This method is called **Static Dictionary Method** and is developed by J.STORER [Ref. 7]. For this method we assume that the message is English text and all the substrings exists in the dictionary. This method is simple and there is no danger that the encoder and decoder dictionaries become different as a result of a noise in communication line, allowing for much simple and efficient error detection and recovery.

Also, in the same area of the dictionary approach exists the **Sliding Dictionary Method** and the **Dynamic Dictionary Method** that update the dictionary during the compression. With these methods it is not necessary to require all the substrings of the original text to exist in the installed dictionary. If a **new** string is recognized by the sender it is transferred and added to the dictionary of the receiver for further use.

III. DICTIONARY DATA COMPRESSION

A. OVERVIEW

The whole idea of this approach is based on the fact that the necessary bits to represent an ASCII character are 8, but with the same number of bits it is possible to represent an integer up to 511. An English word with an average length of 5 letters require $5 \times 8 = 40$ bits to represent using regular ASCII code. However, if each word is represented with an integer number, the required number of bits are only 16 for 65,532 words. This almost covers all common English words used today.

The program developed in this study translates each word into an integer based on a dictionary specified by the user. The program compare the words of the text to be compressed, with the words in the dictionary. The output is a file that contains only numbers. The optimum condition occurs when all the words in the text are in dictionary.

If a word does not exist in the dictionary, the output of the program produces a list of new words used to update the dictionary.

After the creation of the stream of number, the digits of these numbers are compressed using HUFFMAN coding [Ref. 4].

The digits that appeared more frequently in the compressed file will be coded with shorter bit patterns than those with less frequent digits. So the digit 1 and the **BLANK** are coded with only two bits because they appear more frequently than other digits. The other digits are coded with various length bit pattern up to six bits.

The decompression uses an identical dictionary as that used in the compression. Decompression converts the stream of integers into English words. If a word is not in the current dictionary, the program adds the word to the dictionary and then makes the conversion.

The method of updating the dictionary of the receiver is similar to the Dynamic Dictionary Method proposed by J.STORER.
[Ref. 7]

B. COMPRESSION PROGRAM

The program is written in C programming language and consists of three parts: Input, Conversion, and Numerical Translation.

1. Input Part

The input part of the program reads two files: the dictionary and the text that is to be compressed. It creates two linked-lists. Each node in the list contains a word, and a pointer that points to the next node. The function **makelist** reads the files character by character and creates the list of nodes. Two header nodes are created, **headdic** and **headtext**, to

provide the head-and-tail information of the lists as seen in Fig. 9.

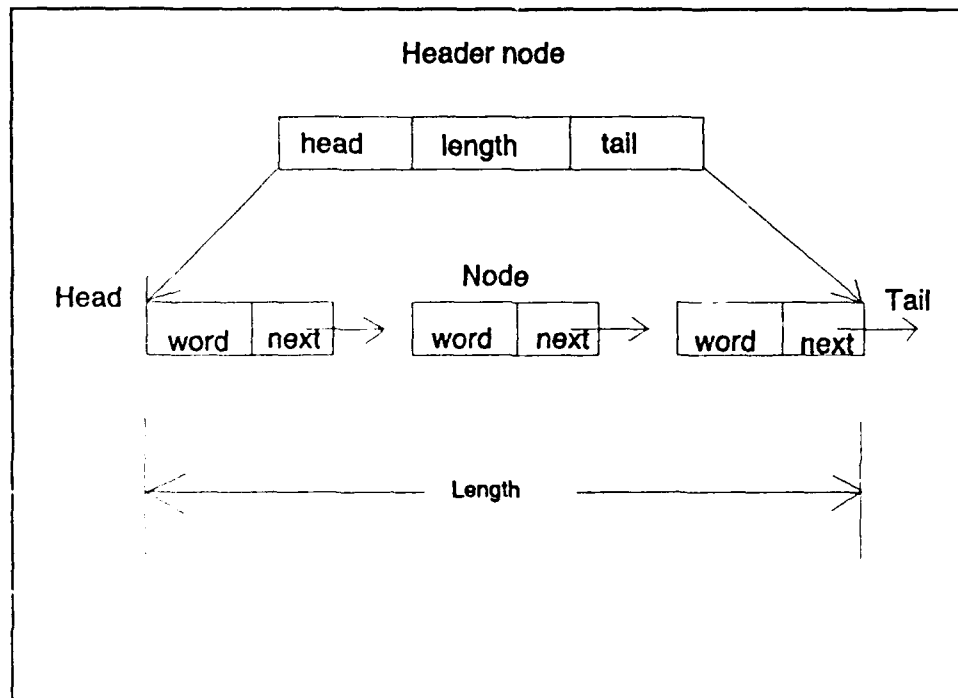


Figure 9 Link list scheme for the Dictionary and Text

2. Conversion Part

This conversion part is the most important of the program because it translates the words to numbers. This task is done by the function **printlist**. This function compares all words of the dictionary with each word of the text. When there is a match, it prints the number of the node in the dictionary. For example, if the word of the text is the same as the third word in the dictionary, then the program converts

the word to the number 3. An example of this appears in Fig. 10.

If the word in the text does not exist in the dictionary the program **creates** a new node in the dictionary list and store the new word for future use. Additionally, the program prints the new word into the compressed file **OUTPUT**. This new word is then used for updating the dictionary of the receiver that will perform the decompression. Usually **OUTPUT** contains a few words at the beginning (new words) and a stream of numbers that is the entire compressed text. The distinction between words and numbers succeed with the symbol σ and is necessary for the decompression program to understand where finished the new words and start the compressed text.

Another task of the function **printlist** is to update the old dictionary, i.e **Diction**. **Diction** is then used for future file compression.

Blanc	1
CR	2
LF	3
.	4
The	5
a	6
is	7
text	8
for	9

Figure 10 Conversion of words into numbers

3. Numerical Translation

In the numerical translation part, there is a function **HOFF_COM** that reads the stream of numbers and counts the frequency of each digit. The function **BIT_COM** then translates each digit into a **Binary Code**.

Each code is unique and recognizable by the decompression program. The codes for each digit are in Fig. 11.

BLANC	00
1	10
2	111
3	0100
4	0110
5	0111
6	1100
7	1101
8	01011
9	010100
0	010101

Figure 11 Bit Conversion of Digits

Finally, all the encoded bits are grouped into bytes before they are sent to the receiver. An example of this conversion and the composition of a character appears in Fig. 12.

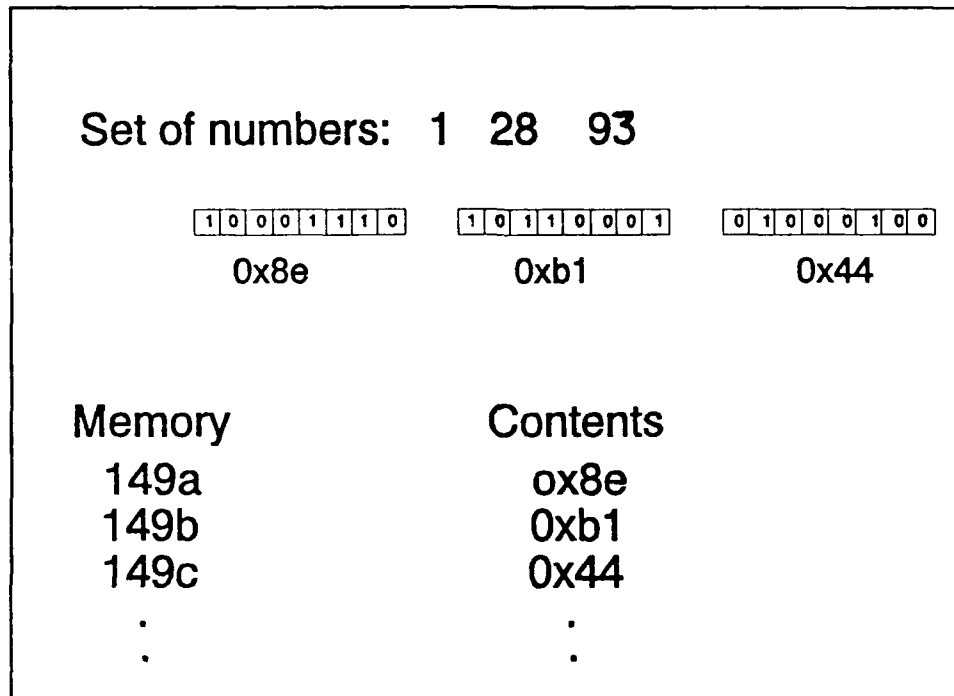


Figure 12 Conversion of numbers and compose of character

Figure 12 shows how the program creates the eight bits characters and stores them in memory. For example consider the set of numbers 1 b 28 b 93.

The program puts into the first two digits of memory 149a, the bit pattern 10 (this comes from table in Fig. 11). After the 1 follows a blank so the program puts into the next two bits, the 00 pattern. Next the 2 is translated into 111. The last bit of this eighth bit pattern is full with the first bit of the pattern of conversion of the number 8 (In this case

the pattern is 01011 so the last bit of the character is 0). This explains why the character 0x8e is stored in the place 149a for this example.

C. DICTIONARY

The dictionary is an important part in the compression process. The running time of the compression program and the compression ratio depend on the way that the dictionary is constructed. The use of an alphabetical dictionary is not a good choice because the program may spend a lot of time comparing the words of the dictionary until reaching the match. Instead, a dictionary based on the usage frequency of each word in the English text is used.

Based on the study of H.KUCERA [Ref. 5] of the most frequently used words today, the top of the dictionary contains words such as the, and a. This allows the program to find the most common words quickly. With this kind of dictionary, the time required for compression is significantly reduced.

Moreover, it is interesting to investigate the number of words that must be contained in the dictionary. As more words are contained in the dictionary, the compression ratio improved because the new words in the compressed file are few (the words occupies a lots of bits). The ideal situation is a dictionary which includes the entire set of the words in the text which yields the maximum compression ratio.

D. DECOMPRESSION PROGRAM

The decompression program is based on three linked-lists. The first one contains the words of the dictionary, the second has the new words that exists in the compressed file called **OUTPUT**, and the third list contains the numbers that are received instead of the whole text.

The words of the second list must be added to the dictionary and the stream of numbers must be converted to the original text.

The first and the second list are read with the function **makelist()** as in the compression program. The third list created with the function **fscanf()** reads from a file that called **intmed** and contains only the numbers that are sent to the transmitter. The nodes of this list are slightly different than the nodes of the previous lists, because instead of the **word** in the node, it contains an integer called **num** and a pointer to the next node. This kind of node is called **numnode**. The **num** in this special node is used for translation of the compressed file into the original one.

The function **printlist()** adds new words to the dictionary that are received with the compressed file, and converts the numbers into English text. The dictionary created from the old one with the addition of the new words called **Dictiona**. The program also contains other functions such as the **NewNode()** which creates the new node for the new word to be added to the

old dictionary, and the function **create()** which creates the first node of each list.

The conversion of the characters that was sent in the compressed file succeeded with the function **Ahoff()**. Actually this function gets character by character from the compressed file and feeds the function **Bit_com()** which makes the conversion into the digits. The **ahoff()** function creates the file **intmed** that contains the stream of numbers necessary for the creation of the **Numlist**.

E. USAGE OF THE PROGRAM

1. Compression

For compression, run the executable file created by the compilation of the **.C** program with two arguments. The first argument is the dictionary and the second is the file to be compressed. The result is a message that the compression complete, and the compressed file called **OUTPUT**. The next time the program is run, it is necessary to use the file **DICTION**, created by the first execution, as the first argument.

2. Decompression

For decompression, the executable file created by the compilation of the **.C** decompression program is used. It also requires two arguments. The first one is the name of the dictionary file (must be the same as that used in the compression part), and the second one is **always** the file **OUTPUT**.

The result is a message that decompression done, and a file that called **ORIGINAL** which is the actual data file that originally compressed. The next time the program is run, **DICTIONA**, which is the up-to-date dictionary, is used as the first argument.

IV. STATISTICAL RESULTS - PERFORMANCE

A. GENERAL

This chapter contains some of the performance results of the dictionary compression scheme. The following tests are made to investigate the performance of the method.

1. Size of file vs. compression ratio
2. Comparison between different compression schemes

B. SIZE OF FILE VS. COMPRESSION RATIO

In this test, different file sizes were examined. The results are plotted in Fig. 13. This figure exhibits a family of curves, each representing a different percentage of existence in the dictionary.

Curve A is the case where all the words of the text exist in the dictionary. Curve B represents the case where only 75% of the text words exist in the dictionary, curve C is the 50% case and D curve is the case for 25%.

The vertical axis is the compression ratio and the horizontal axis the size of files in bytes. The horizontal axis is in log and the vertical is linear. The compression ratios is calculated from the formula:

$$\text{Comp.Ratio} = (1 - \text{Comp.File}/\text{Origin.File}) * 100$$

Another important parameter in these curves is the size of the dictionary. Fig. 13 is based in a dictionary with less

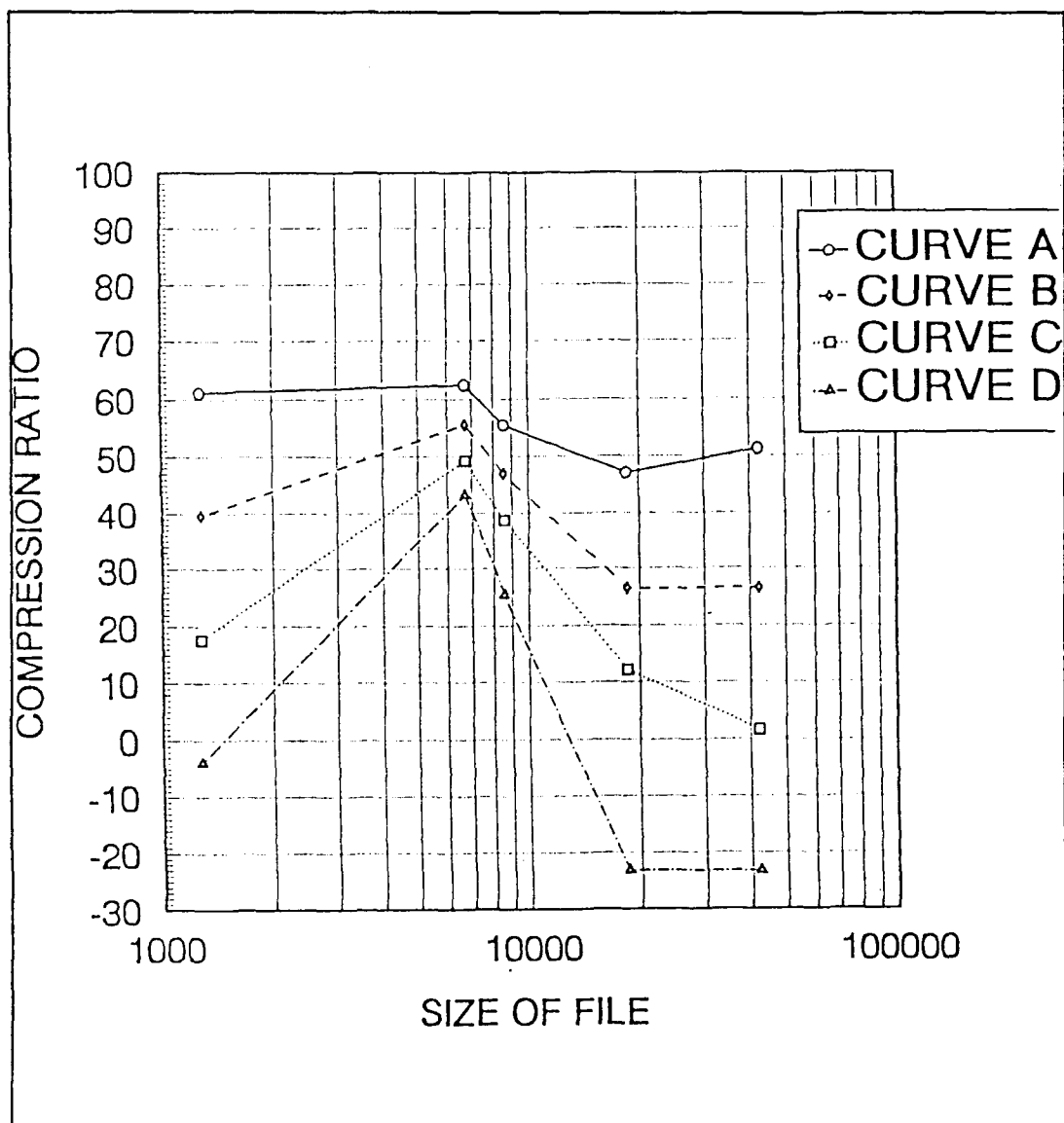


Figure 13 Compression ratio vs. size of file with dictionary with less than 1000 words

than 1000 words.

Curve A represents the highest compression ratio, which is expected since all the words exist in the dictionary and the compressed file contains only numbers. As we move to the curves B, C, and D, the compression is reduced and sometimes even exhibit **negative** values. This means that instead of

compression the program provides **expansion**, i.e, the **OUTPUT** file is bigger than the **ORIGINAL** text. This may occur for small files (less than 2K) and also larger files (more than 40k).

The explanation of this phenomenon is that in small files when the dictionary has only **25%** known words the program must send the rest words of the text like new words, so the output file is bigger than the original one, hence the compression ratio is **negative**. The same phenomenon happens with big files.

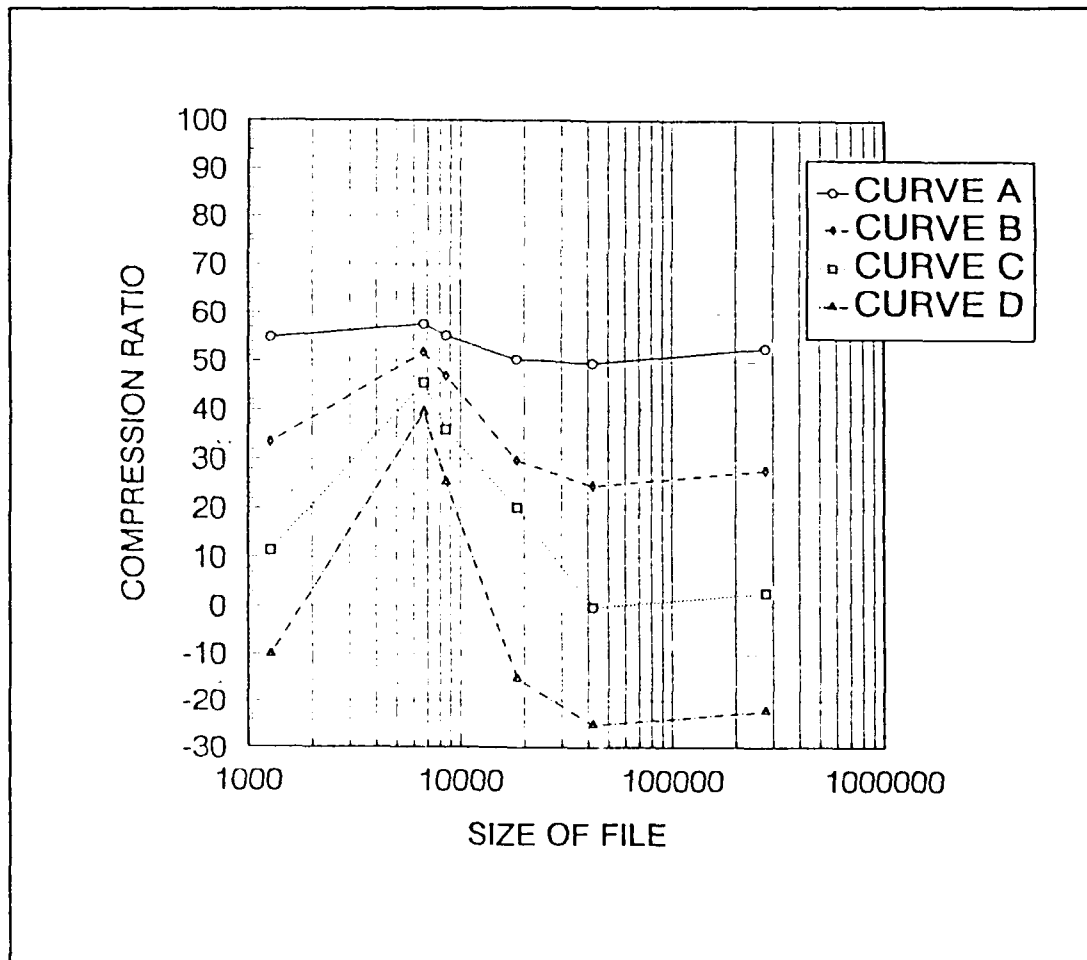


Figure 14 Compression ratio vs. size file with dictionary with more than 3000 words

Figure 14 shows the same family of curves as those of Fig. 13 but the dictionary contains more than 3000 words. The difference between Figure 13 and 14 is that now we must send larger numbers for each word and the compression ratio is slightly smaller than those in Fig. 13.

C. COMPARISON BETWEEN DIFFERENT COMPRESSION SCHEMES

Figure 15 shows the curves of the compression ratio between three compression schemes: The **COMPRESS** of UNIX [Ref. 10], the **Stacpack** of Stac.INC (commercial program) [Ref. 8] and the **Dictionary** scheme developed in this study.

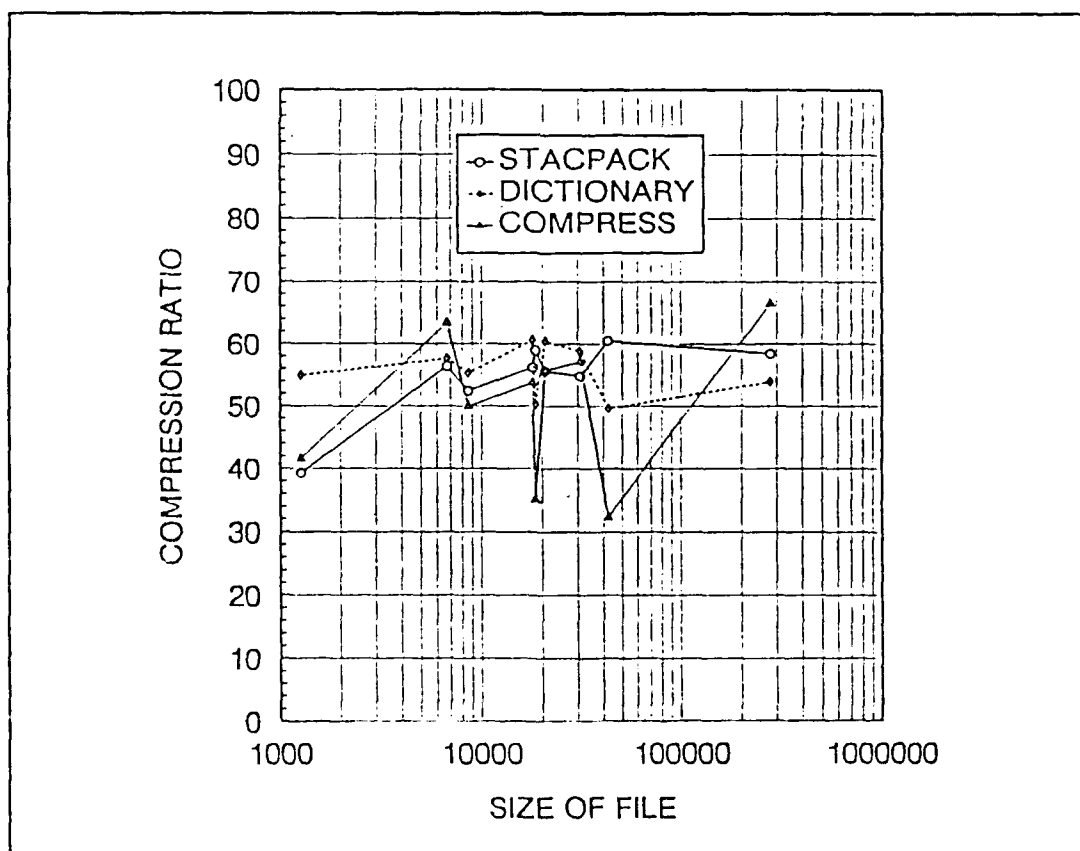


Figure 15 Comparison between the three schemes

The horizontal axis is the size of the file in log and the vertical axis is the compression ratio. The data for the dictionary comes from a dictionary of more than 3000 words. The dictionary contains all the words of the text.

D. NON-REVERSIBLE VERSION OF THE DICTIONARY PROGRAM

So far we have developed reversible one algorithm. It is interesting to see the algorithm that is not reversible. This algorithm does not **count BLANK** spaces in the text. Any numbers of consecutive blanks are represented as one single blank. Since this algorithm is not reversible we can significantly increase the compression ratio as shown in Figures 16 and 17 for two different kinds of dictionaries.

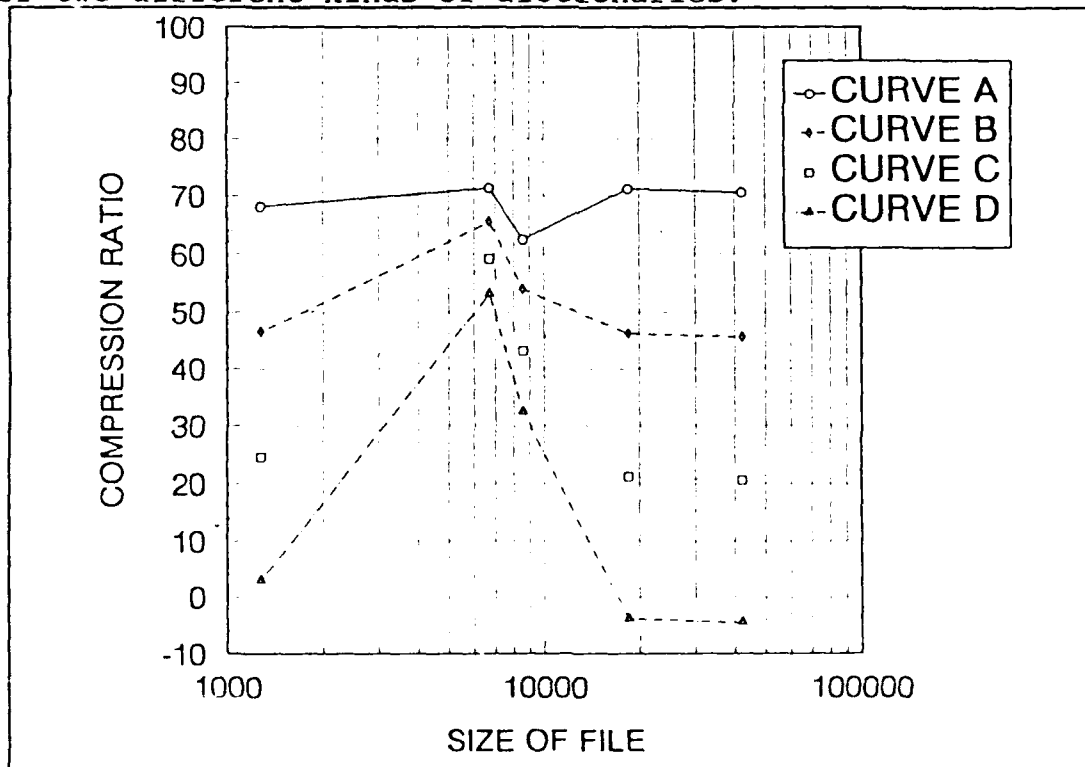


Figure 16 Compression ratio vs. size of file with dictionary with less of 1000 words for non-reversible algorithm

The curves A, B, C, and D are for dictionaries that contains 100%, 75%, 50%, and 25% percentage of words, respectively. Fig. 16 is for a dictionary with less than 1000 words and the Fig. 17 is for a dictionary with more than 3000 words.

This kind of algorithm (don't-care-blank) could be used in cases where blank spaces inside the text can be neglected. For example, instead of 5 blank spaces in a row in the original text the algorithm recovers only one. This algorithm gives a better compression ratio than the reversible version.

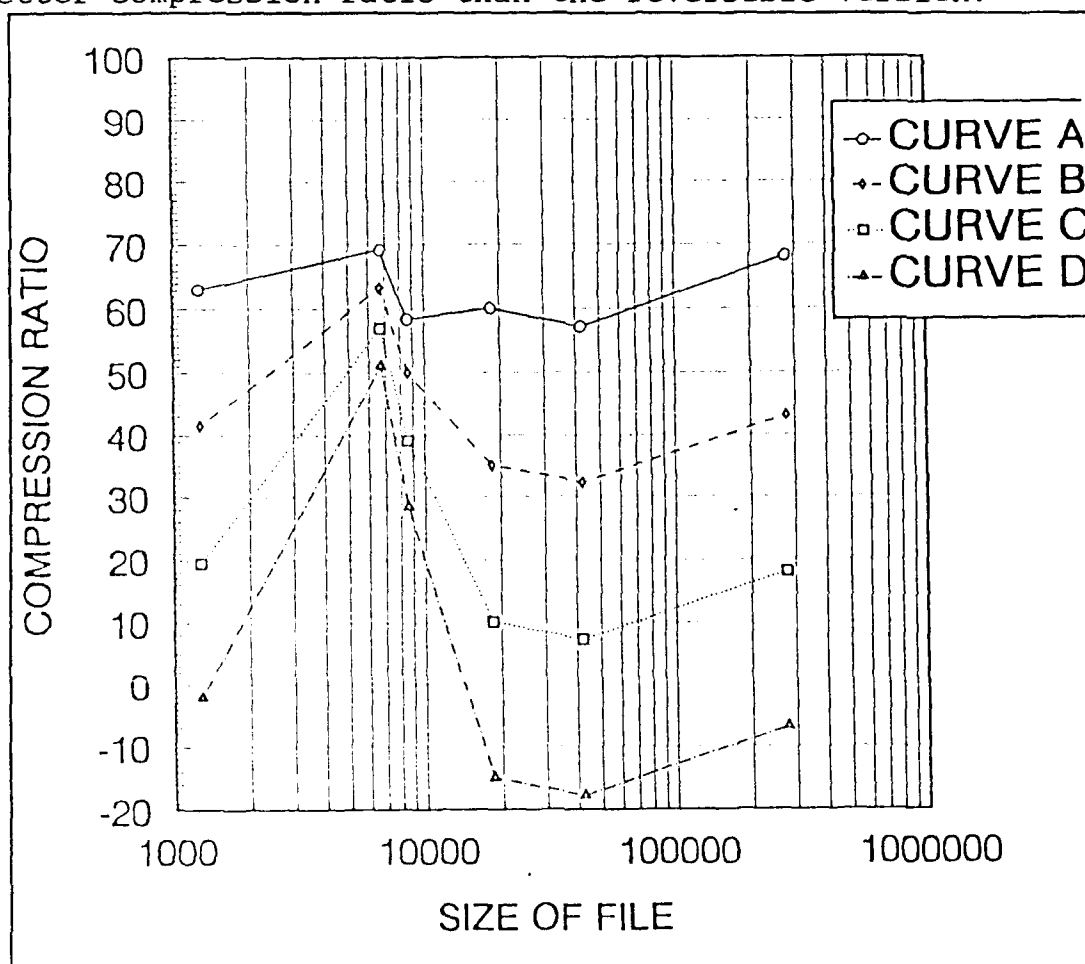


Figure 17 Compression ratio vs. size of file for dictionary with more than 3000 words for non-reversible algorithm

The results of the comparison between the non-reversible algorithm and the other compression programs is shown in Fig. 18. The horizontal axis is the size of the file in Kbytes and the vertical the compression ratio.

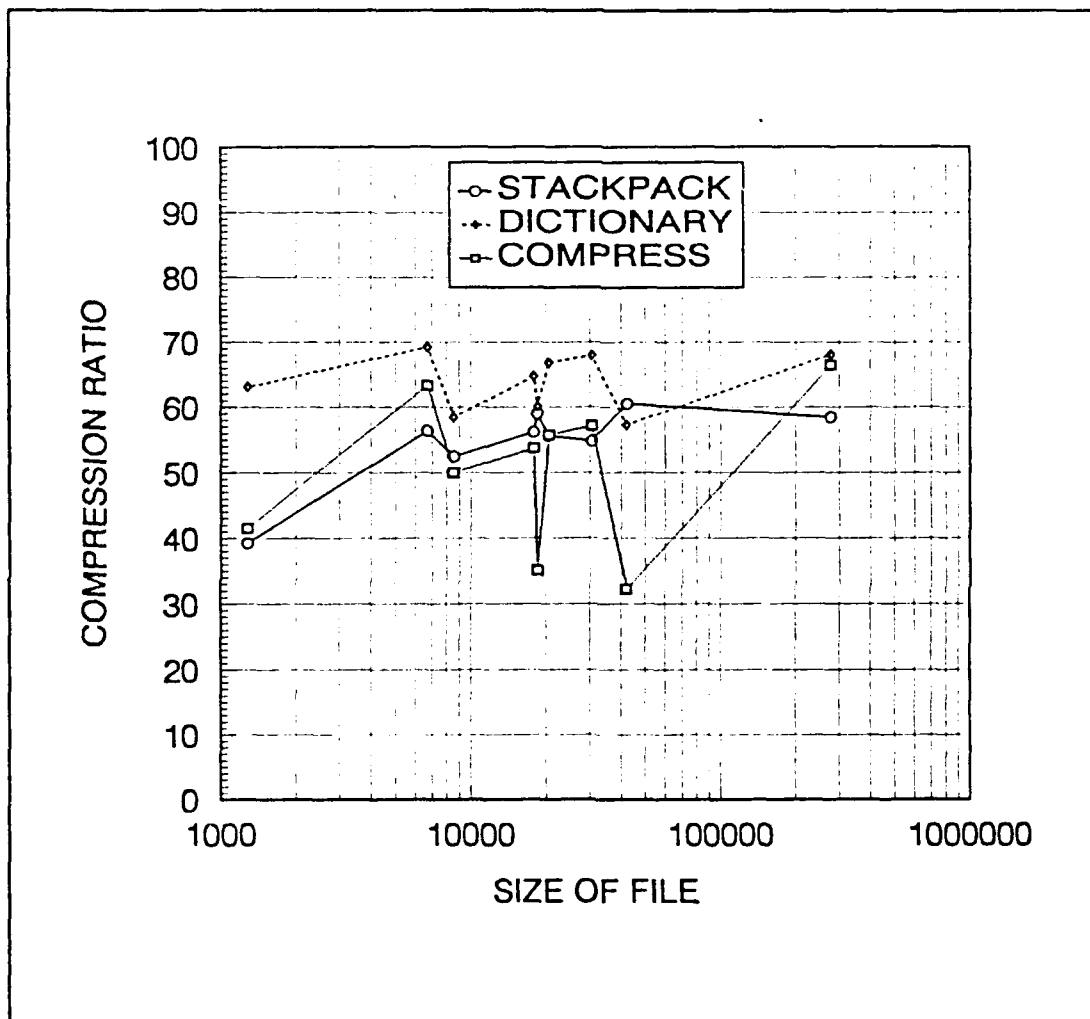


Figure 18 Comparison between the non-reversible algorithm and the others compression schemes

E. TIME IMPROVEMENT ALGORITHM

The main problem of the developed algorithm is the time that spent the program searching the whole list of the dictionary for matching with the words of the text. A version of the algorithm that gives solution on this problem is in the appendix A.

The solution is that instead of singly linked-list for the dictionary the improved algorithm has 28 separate lists. The 26 are for the lower case letters (one for each letter), the 27th is for all the punctuation symbols like CR, SP, LF and the 28th is for all the capital letters. In addition the structure of each node contains a field, called **val** an integer that is used to indicate the place that word is ordered in the dictionary.

The program reads the dictionary and place each word and its number in the proper list. Then the program starts reading the text file to be compressed. It checks the first symbol of each word, and start searching for match in the list that contains this symbol. If it finds the same word it then prints the number that represent this word in the dictionary. In case that the word is **new** it adds the word in the list and print a number that is larger than the last number of the dictionary by one.

The rest of the algorithm about the numerical translation remains the same. With this kind of algorithm the whole execution time may be improved 27 times.

The algorithm could be further improved if instead of 28 list we may use 53 or more. For example we may separate capital symbols, that is to have one list for each capital letter. In addition the punctuation symbols may be subdivided.

V. CONCLUSIONS

A. DATA COMPRESSION

The programs provided within Appendix A achieve the primary goal of this thesis. The goal was to develop a new data compression scheme different from those already existing, with better results. This goal was met through the dictionary approach and numerical translation. Algorithms for compression and decompression were developed and implemented. Implementation and successful testing of the algorithm verify the accomplishment of the primary goal.

B. FUTURE RESEARCH

The opportunity exists for improving the developed scheme not in the area of compression ratio, but rather in the area of time. This could be done with a different organization of the dictionary and the text. Instead of using multiple linked lists for the dictionary, it could be organized in some kind of tree so it is possible to further reduce the searching time for matching.

APPENDIX A: PROGRAM LISTINGS

A. COMPRESSION PROGRAM

```
/* This is the program that makes the COMPRESSION. */
/* It is the complete reversible algorithm */

#include <stdio.h>
#include <string.h>
#include <malloc.h>

#define MAXLEN      25
#define MALLOC(x)   ((x *) malloc(sizeof(x)))
#define PUNCT(x)    ((x=='\n') || (x=='\t') || (x=='\r') || (' '<=x
&& x<='/' ) || (':'<=x && x<='@') || ('['<=x && x<='`') || ('{'<=x &&
x<='~'))
#define RECOG(x)    (x=='^A')

struct node
{
    char      *word;
    struct node *next ;
};
typedef struct node_type;

struct header
{
    int      length;
    node_type *head, *tail;
};
typedef struct header head_type;

head_type      *header_new, *headdic, *headtext;
FILE           *fopen(), *fp, *fout, *fdic, *fhoff, *fred,
*ftel, *fend;
FILE           *ffin, *fwor, *fnum;
char           *lala;

void           print_list();
void           hoff_com();
void           bit_com();
void           connect();

main (argc, argv)
```

```

int          argc;
char         *argv[];
{
    head_type  *create(), *makelist(), *makelist1();

    headdic = makelist(argv[1]);
    headtext = makelist1(argv[2]);

    print_list();
    hoff_com();
    connect();

    printf("\n\n    Compressssion done!");
    printf("\n\n    The compressed file called OUTPUT");

    exit(0);
}

head_type *makelist(filein)
char      *filein;
{
    register int  c, length;
    node_type     *new, *NewNode();
    char          buffer[MAXLEN + 1];
    head_type     *headd;
    head_type     *create();
    int           p;

    headd = create();
    if ((fp = fopen(filein, "r")) == NULL)
    {
        printf("ERROR ! I can't open %s\n", filein);
        exit(0);
    }
    strcpy(buffer, "      ");
    length = 0;
    for(;;)
    {
        c = getc(fp);
        if (c == EOF) break;
        p = 1;
        if ((060<=c && c<=071) || (0101<=c && c<=0132) || (0141<=c
&& c<=0172))
        {
            buffer[length++] = c;
        }
        if ((PUNCT(c)) && (buffer[0] != ' '))
        {
            buffer[length] = '\\0';
            new = NewNode();
            new->next = NULL;

```

```

        if (headd->length == 0)
        {
            headd->head = new;
        }
        else
        {
            (headd->tail)->next = new;
        }
        headd->tail = new;
        headd->length++;
        new->word = (char
*)malloc(sizeof(char)*strlen(buffer) +1);
        strcpy(new->word, buffer);
        strcpy(buffer, " ");
        length = 0;
        p = 0;
    }
    if ((PUNCT(c)) && (p == 1))
    {
        buffer[0] = c;
        buffer[1] = '\0';
        new = NewNode();
        new->next = NULL;
        if (headd->length == 0)
        {
            headd->head = new;
        }
        else
        {
            (headd->tail)->next = new;
        }
        headd->tail = new;
        headd->length++;
        new->word = (char
*)malloc(sizeof(char)*strlen(buffer) +1);
        strcpy(new->word, buffer);
        strcpy(buffer, " ");
        length = 0;
        c = getc(fp);
    }
} /* end for */
return (headd);
}
head_type *makelist1(filein)
char *filein;
{
    register int c, length;
    node_type *new, *NewNode();
    char buffer[MAXLEN + 1];
    head_type *headd;

```

```

head_type      *create();

headd = create();
if ((fp = fopen(filein,"r")) == NULL)
{
    printf("ERROR ! I can't open %s\n",filein);
    exit(0);
}
strcpy(buffer, "      ");
length = 0;
for(;;)
{
    c = getc(fp);
    if (c == EOF) break;
    if ((060<=c && c<=071) || (0101<=c && c<=0132) || (0141<=c
&& c<=0172))
    {
        buffer[length++] = c;
    }
    if ((PUNCT(c)) && (buffer[0] != ' '))
    {
        buffer[length]= '\0';
        new = NewNode();
        new->next = NULL;
        if (headd->length == 0)
        {
            headd->head = new;
        }
        else
        {
            (headd->tail)->next = new;
        }
        headd->tail = new;
        headd->length++;
        n e w - > w o r d      =      ( c h a r
*)malloc(sizeof(char)*strlen(buffer) +1);
        strcpy(new->word, buffer);
        strcpy(buffer, "      ");
        length = 0;
    }
    if (PUNCT(c))
    {
        buffer[0] = c;
        buffer[1]= '\0';
        new = NewNode();
        new->next = NULL;
        if (headd->length == 0)
        {
            headd->head = new;
        }
        else
    }
}

```



```

        {
            (headd->tail)->next = new;
        }
        headd->tail = new;
        headd->length++;
        new->word = (char
*)malloc(sizeof(char)*strlen(buffer) +1);
        strcpy(new->word, buffer);
        strcpy(buffer, " ");
        length = 0;
    }
    /* end for */
    return (headd);
}
node_type *NewNode()
{
    node_type *newnode;
    if (!(newnode = MALLOC(node_type)))
    {
        printf(" Out of storage \n");
        exit(1);
    }
    return(newnode);
}

head_type *create()
{
    if (header_new = MALLOC(head_type))
    {
        header_new->length = 0;
        header_new->head = header_new->tail = NULL;
    }
    return(header_new);
}

void print_list()
{
    int r, k, m;
    node_type *dicptr, *textptr, *mew;

    m = 1;
    k = headdic->length;
    fout = fopen("wordout", "w+");
    textptr = headtext->head;
    for ( ; textptr != NULL; )
    {
        dicptr = headdic->head;
        for ( ; dicptr != NULL; )
        {
            if (strcmp((textptr->word), (dicptr->word)) ==
0) break;

```

```

        dicptr=dicptr->next;
    }
    if (dicptr == NULL) /* end of dic list? if so, add
it now */
    {
        mew = NewNode();
        mew->next = NULL;
        mew->word = (char
*)malloc(sizeof(char)*strlen(textptr->word)+1);
        strcpy((mew->word),(textptr->word));
        (headdic->tail)->next = mew; /* new tail */
        headdic->tail = mew;
        headdic->length++;
        fprintf(fout,"%s ",mew->word);
        printf("%s ",mew->word);
    }
    textptr = textptr->next;
}
fprintf(fout,"^A");
printf ("^A");
fclose(fout);
fhoff = fopen("hoffman","w+");
textptr = headtext->head;
for (; textptr != NULL; )
{
    r=1;
    dicptr = headdic->head;
    for (; dicptr != NULL; )
    {
        if (strcmp((textptr->word),(dicptr->word)) ==
0)
        {
            fprintf(fhoff,"%d ",r);
            printf("%d ",r);
            break;
        }
        r++;
        dicptr = dicptr->next;
    }
    textptr = textptr->next;
}
fprintf(fhoff," ");
fclose(fhoff);
fdic = fopen("diction","w+");
dicptr = headdic->head;
for ( ; dicptr != NULL; )
{
    fprintf(fdic,"%s\n",dicptr->word);
    dicptr = dicptr->next;
}
fclose(fdic);

```

```

}

void      hoff_com()
{
    char          c;
    static int     counter, tempa;
    int           k, s, totbit, siz ;
    int           b=0, a0=0, a1=0, a2=0, a3=0, a4=0, a5=0,
a6=0, a7=0, a8=0, a9=0;
    float         pb=0.0, p0=0.0, p1=0.0, p2=0.0, p3=0.0,
p4=0.0;
    float         p5=0.0, p6=0.0, p7=0.0, p8=0.0, p9= 0.0;

    fred = fopen("hoffman","r+");
    for (; ;)
    {
        c = getc(fred);
        if (c == EOF) break;
        if ( c == ' ' ) b++;
        if ( c == '0' ) a0++;
        if ( c == '1' ) a1++;
        if ( c == '2' ) a2++;
        if ( c == '3' ) a3++;
        if ( c == '4' ) a4++;
        if ( c == '5' ) a5++;
        if ( c == '6' ) a6++;
        if ( c == '7' ) a7++;
        if ( c == '8' ) a8++;
        if ( c == '9' ) a9++;
    }
    fclose(fred);
    s = (b+a0+a1+a2+a3+a4+a5+a6+a7+a8+a9);
    totbit = (b*2 + a0*6 + a1*2 + a2*3 + a3*4 + a4*4 + a5*4
              + a6*4 + a7*4 +a8*5 + a9*6);
    printf ("Total bits: %d\n",totbit);
    pb = (float)(b)/(float)(s)*100;
    p0 = (float)(a0)/(float)(s)*100;
    p1 = (float)(a1)/(float)(s)*100;
    p2 = (float)(a2)/(float)(s)*100;
    p3 = (float)(a3)/(float)(s)*100;
    p4 = (float)(a4)/(float)(s)*100;
    p5 = (float)(a5)/(float)(s)*100;
    p6 = (float)(a6)/(float)(s)*100;
    p7 = (float)(a7)/(float)(s)*100;
    p8 = (float)(a8)/(float)(s)*100;
    p9 = (float)(a9)/(float)(s)*100;

    siz = totbit/8 + 1 ;
    printf("Siz %d\n",siz);
    lala = (char *) malloc(siz * sizeof(char));
    fred = fopen("hoffman","r+");

```

```

tempa = 0x0000;
counter = 0;
fend = fopen("teliko","w+");
for (; ;)
{
    c = getc(fred);
    if (c == EOF) break;
    bit_com(c);
}
fclose(fend);
fclose(fred);
}

void    bit_com(number)
char    number;
{
    static int    tempa, counter;
    int    mask = 0x0000, temp2 = 0x0000;
    int    cormask, bhof, m;
    int    acounter;

    switch (number)
    {
        case ' ':
            bhof = 0x0000;
            acounter = 14;
            bhof <=< acounter;
            bhof >>= counter;
            tempa = tempa | bhof;
            counter = counter + 2;
            break;

        case '1':
            bhof = 0x0002;
            acounter = 14;
            bhof <=< acounter;
            bhof >>= counter;
            if (counter != 0)
            {
                cormask = 0x8000;
                for ( m = 1; m < counter; m++)
                {
                    cormask >>= 1;
                }
                bhof = bhof ^ cormask;
            }
            tempa = tempa | bhof;
            counter = counter + 2;
            break;

        case '8':

```

```

        bhof = 0x000b;
        acounter = 11;
        bhof <=<= acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 5;
        break;

    case '2':
        bhof = 0x0007;
        acounter = 13;
        bhof <=<= acounter;
        bhof >>= counter;
        if (counter != 0)
        {
            cormask = 0x8000;
            for ( m = 1; m < counter; m++)
            {
                cormask >>= 1;
            }
            bhof = bhof ^ cormask;
        }
        tempa = tempa | bhof;
        counter = counter + 3;
        break;

    case '7':
        bhof = 0x000d;
        acounter = 12;
        bhof <=<= acounter;
        bhof >>= counter;
        if (counter != 0)
        {
            cormask = 0x8000;
            for ( m = 1; m < counter; m++)
            {
                cormask >>= 1;
            }
            bhof = bhof ^ cormask;
        }
        tempa = tempa | bhof;
        counter = counter + 4;
        break;

    case '3':
        bhof = 0x0004;
        acounter = 12;
        bhof <=<= acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 4;

```

```

        break;

case '6':
    bhof = 0x000c;
    acounter = 12;
    bhof <<= acounter;
    bhof >>= counter;
    if (counter != 0)
    {
        cormask = 0x8000;
        for ( m = 1; m < counter; m++)
        {
            cormask >>= 1;
        }
        bhof = bhof ^ cormask;
    }
    tempa = tempa | bhof;
    counter = counter + 4;
    break;

case '4':
    bhof = 0x0006;
    acounter = 12;
    bhof <<= acounter;
    bhof >>= counter;
    tempa = tempa | bhof;
    counter = counter + 4;
    break;

case '9':
    bhof = 0x0014;
    acounter = 10;
    bhof <<= acounter;
    bhof >>= counter;
    tempa = tempa | bhof;
    counter = counter + 6;
    break;

case '5':
    bhof = 0x0007;
    acounter = 12;
    bhof <<= acounter;
    bhof >>= counter;
    tempa = tempa | bhof;
    counter = counter + 4;
    break;

case '0':
    bhof = 0x0015;
    acounter = 10;
    bhof <<= acounter;

```

```

        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 6;
        break;
    }
    if ( counter >= 8 )
    {
        temp2 = tempa & 0xff00;
        temp2 >>= 8;
        if (temp2 == 0x1a) temp2 = 0x1;
        if (temp2 == 0xffff) temp2 = 0x81;
        *lala = temp2;
        fprintf(fend,"%c",*lala);
        lala++;
        tempa <<= 8;
        counter = counter - 8;
    }
}

void connect()
{
    char      c, d;

    ffin = fopen("output","w+");
    fwor = fopen("wordout","r+");
    fnum = fopen("teliko","r+");

    for (;;)
    {
        c = getc(fwor);
        if ( c == EOF) break;
        putc(c,ffin);
    }
    for (;;)
    {
        d = getc(fnum);
        if ( d == EOF) break;
        putc(d,ffin);
    }
    fclose(ffin);
    fclose(fwor);
    fclose(fnum);
}

```

B. DECOMPRESSION PROGRAM

```
/* This is the program that makes the DECOMPRESSION. */

#include <stdio.h>
#include <string.h>
#include <malloc.h>

#define MAXLEN      25
#define MALLOC(x)   ((x *) malloc(sizeof(x)))
#define PUNCT(x)    ((x=='\n') || (x=='\t') || (x=='\r') || (' '<=x
&& x<='/') || (':'<=x && x<='@') || ('['<=x && x<='`') || ('{'<=x &&
x<='}') )
#define RECOG(x)    (x=='^A')

struct node
{
    char      *word;
    struct node *next ;
};
typedef struct node_type;

struct numnode
{
    int      num;
    struct numnode *numnext ;
};
typedef struct numnode_type;

struct header
{
    int      length;
    node_type *head, *tail;
};
typedef struct header head_type;

struct numheader
{
    int      length;
    numnode_type *numhead, *numtail;
};
typedef struct numheader numhead_type;

head_type      *header_new, *headdic, *headtext;
numhead_type   *header_numnew, *headnum;
FILE           *fopen(), *fp, *fp1, *fp2, *fp3, *fdic, *fori,
               *fred, *freg;
```



```

void          print_list();
void          ahoff();
void          bit_com();

main (argc, argv)
int          argc;
char         *argv[];
{
    int          m = 0, puffer, length;
    head_type    *create(), *makelist();
    numhead_type *numcreate();
    node_type    *listext;
    numnode_type *vew, *NumNewNode();
    char         c;

    headdic = makelist(argv[1]);
    headtext = makelist(argv[2]);
    fp1 = fopen("codhof", "w+");
    fp3 = fopen(argv[2], "r");
    for(;;)
    {
        c = getc(fp3);
        if ( c == EOF )
        {
            break;
        }
        if ( m != 0 )
        {
            putc(c, fp1);
            m++;
        }
        if( RECOG(c) ) m = 1;
    }
    fclose(fp3);
    fclose(fp1);

    ahoff();

    fp2 = fopen("intmed", "r+");
    headnum = numcreate();
    length = 0;
    while (fscanf(fp2, "%d", &puffer) != EOF)
    {
        vew = NumNewNode();
        vew->numnext = NULL;
        if (headnum->length == 0)
        {
            headnum->numhead = vew;
        }
        else
        {

```

```

        (headnum->numtail)->numnext = vew;
    }
    headnum->numtail = vew;
    headnum->length++;
    (vew->num) = puffer;
}
fclose(fp2);

print_list();

printf("\n\n    Decompression done!");
printf("\n\n    The decompressed file called ORIGINAL");

exit(0);
}

head_type *makelist(filein)
char      *filein;
{
    register int    c, length;
    node_type      *new, *NewNode();
    char           buffer[MAXLEN + 1];
    head_type      *headd;
    head_type      *create();
    int            p;

    headd = create();
    if ((fp = fopen(filein,"r")) == NULL)
    {
        printf("ERROR ! I can't open %s\n",filein);
        exit(0);
    }
    strcpy(buffer, "      ");
    length = 0;
    for(;;)
    {
        c = getc(fp);
        p = 1;
        if (c == EOF || c == '^A')
        {
            break;
        }
        if ((060<=c && c<=071) || (0101<=c && c<=0132) || (0141<=c
            && c<=0172))
        {
            buffer[length++] = c;
        }
        if (PUNCT(c) && (buffer[0] != ' '))
        {
            buffer[length]= '\0';
            new = NewNode();

```

```

new->next = NULL;
if (headd->length == 0)
{
    headd->head = new;
}
else
{
    (headd->tail)->next = new;
}
headd->tail = new;
headd->length++;
new->word = (char
            *)malloc(sizeof(char)*strlen(buffer) +1);
strcpy(new->word, buffer);
strcpy(buffer, "      " );
length = 0;
p = 0;
}
if (PUNCT(c) && (p == 1))
{
    buffer[0] = c;
    buffer[1]= '\0';
    new = NewNode();
    new->next = NULL;
    if (headd->length == 0)
    {
        headd->head = new;
    }
    else
    {
        (headd->tail)->next = new;
    }
    headd->tail = new;
    headd->length++;
    new->word = (char
                *)malloc(sizeof(char)*strlen(buffer) +1);
    strcpy(new->word, buffer);
    strcpy(buffer, "      " );
    length = 0;
    c =getc(fp);
}
} /* end for */
return (headd);
}

node_type *NewNode()
{
    node_type *newnode;
    if (!(newnode = MALLOC(node_type)))
    {
        printf(" out of storage mike \n");
    }
}

```

```

        exit(1);
    }
    return(newnode);
}

numnode_type *NumNewNode()
{
    numnode_type *numnewnode;
    if (!(numnewnode = MALLOC(numnode_type)))
    {
        printf(" out of storage \n");
        exit(1);
    }
    return(numnewnode);
}

head_type *create()
{
    if (header_new = MALLOC(head_type))
    {
        header_new->length = 0;
        header_new->head = header_new->tail = NULL;
    }
    return(header_new);
}

numhead_type *numcreate()
{
    if (header_numnew = MALLOC(numhead_type))
    {
        header_numnew->length = 0;
        header_numnew->numhead = header_numnew->numtail =
NULL;
    }
    return(header_numnew);
}

void print_list()
{
    int i;
    node_type *dicptr, *textptr, *mew;
    numnode_type *numptr;

    textptr = headtext->head;
    for ( ; textptr != NULL; )
    {
        /* if (strcmp(textptr->word," \0") == 0 )
        {
            textptr = textptr->next;
            if ( textptr == NULL ) break;
        }
        */
    }
}

```

```

        dicptr = headdic->tail;
        mew = NewNode();
        mew->next = NULL;
        (mew->word) = (char *) malloc(sizeof(char) *
strlen(textptr->word)+1);
        strcpy((mew->word), (textptr->word));
        (headdic->tail)->next = mew;
        (headdic->tail) = mew;
        headdic->length++;
        textptr = textptr->next;
    }
    numptr = headnum->numhead;
    fori = fopen("original","w+");
    for ( ; numptr != NULL; )
    {
        dicptr = headdic->head;
        for ( r = 1; r <= headdic->length; r++ )
        {
            if ((numptr->num) == r )
            {
                fprintf(fori,"%s",dicptr->word);
                printf("%s",dicptr->word);
                break;
            }
            dicptr = dicptr->next;
        }
        numptr = numptr->numnext;
    }
    fclose(fori);

    fdic = fopen("dictiona","w+");
    dicptr = headdic->head;
    for ( ; dicptr != NULL; )
    {
        fprintf(fdic,"%s\n",dicptr->word);
        dicptr = dicptr->next;
    }
    fclose(fdic);
}

/* This the decompression programm for the hoffman */

void      ahoff()
{
    char          c;
    static int     counter, tempa;
    int           k, met, s, totbit, siz ;

    fred = fopen("codhof","r+");
    freg = fopen("intmed","w ");
    for ( ; ; )

```

```

    {
        c = getc(fred);
        if (c == EOF) break;
        if (c == 0x1) c = 0x1a;
        if (c == 0xff81) c = 0x00ff;
        bit_com(c);
    }
    fclose(freg);
    fclose(fred);
}

void    bit_com(number)
char    number;
{
    char        temp2;
    static char    tempa, counter = 0;
    int         mask = 0x0080, m;

    for ( m = 1; m <= 8; m++)
    {
        temp2 = number & mask;
        mask = mask/2;
        /*      printf ("temp2 %x mask %x\n",temp2, mask);*/
        counter++;
        tempa <<= 1;
        if ( temp2 != 0) tempa = tempa +1;
        temp2 = 0x0;
        /*      printf("tempa %x counter %d\n",tempa, counter); */
        if ( tempa == 0x0 && counter == 2)
        {
            printf(" ");
            fprintf(freg," ");
            tempa = 0x0000;
            counter = 0;
        }
        if ( tempa == 0x2 && counter == 2)
        {
            printf("1");
            fprintf(freg,"1");
            tempa = 0x0000;
            counter = 0;
        }

        if ( tempa == 0xb && counter == 5 )
        {
            printf("8");
            fprintf(freg,"8");
            tempa = 0x0000;
            counter = 0;
        }
        if ( tempa == 0x7 && counter == 3 )

```

```

    {
        printf("2");
        fprintf(freg,"2");
        tempa = 0x0000;
        counter = 0;
    }
if ( tempa == 0xd && counter == 4 )
{
    printf("7");
    fprintf(freg,"7");
    tempa = 0x0000;
    counter = 0;
}
if ( tempa == 0x4 && counter == 4 )
{
    printf("3");
    fprintf(freg,"3");
    tempa = 0x0000;
    counter = 0;
}
if ( tempa == 0xc && counter == 4 )
{
    printf("6");
    fprintf(freg,"6");
    tempa = 0x0000;
    counter = 0;
}
if ( tempa == 0x6 && counter == 4 )
{
    printf("4");
    fprintf(freg,"4");
    tempa = 0x0000;
    counter = 0;
}
if ( tempa == 0x14 && counter == 6 )
{
    printf("9");
    fprintf(freg,"9");
    tempa = 0x0000;
    counter = 0;
}
if ( tempa == 0x7 && counter == 4 )
{
    printf("5");
    fprintf(freg,"5");
    tempa = 0x0000;
    counter = 0;
}
if ( tempa == 0x15 && counter == 6 )
{
    printf("0");

```

```

        fprintf(freg, "0");
        tempa = 0x000;
        counter = 0;
    }
}

```

C. TIME IMPROVEMENT ALGORITHM

```

/* This is the time improved program that makes the
   COMPRESSION. */
/* It is the complete reversible algorithm */

#include <stdio.h>
#include <string.h>
#include <malloc.h>

#define MAXLEN 15
#define Nlist 29
#define TRUE 1
#define FALSE 0
#define PUNCT(x) ((x=='\n') || (x=='\t') || (x=='\r') || (' ' <= x &&
x <= '/') || (':' <= x && x <= '@') || ('[' <= x && x <= '`') || ('{' <= x &&
x <= '~'))

FILE *fid, *fword, *fhoff, *fdic, *fred, *fend, *ffin, *fwor,
*fnum;
char *lala;

struct node {
    int val;
    char *word;
    struct node *next;
};

struct Hnode {
    int length;
    struct node *head, *tail;
};

/* list[] are head pointers to each list */
void hoff_com();
void bit_com();
void connect();

main(argc, argv)
int argc;
char **argv;
{

```



```

struct node *new, *newnode();
char buffer[MAXLEN];
struct Hnode *list[Nlist];
int i, index, ctr, v;
int j, FOUND, k, c, p, leng;
struct node *ptr;

for(i=0; i<Nlist; i++)
{
    if(list[i]=(struct Hnode *) malloc(sizeof(struct
Hnode)))
    { list[i]->length=0;
      list[i]->head=list[i]->tail=NULL;}
    }
ctr=1;
v = 0;
fid=fopen(argv[1],"r");
strcpy(buffer, " ");
leng = 0;
for(;;) {
    c = getc(fid);
    if (c == EOF ) break;
    p =1;
    if (v == 0)
    {
        if( 65< (int) c && (int) c <90) index = 28;
        else{
            if((int) c < 97 || (int) c > 122) index=27;
            else index = (int) c - 97;
        }
    }
    if ((060<=c && c<=071)|| (0101<=c && c<=0132)|| (0141<=c
&& c<=0172))
    {
        buffer[leng++] = c;
        v= 1;
    }

    if ((PUNCT(c)) && (buffer[0] != ' '))
    {
        buffer[leng] = '\0';
        new=newnode();
        new->val=ctr;
        n e w ~ > w o r d = ( c h a r
*)malloc(sizeof(char)*(strlen(buffer)+1));
        strcpy(new->word, buffer);
        if(list[index]->length==0) {/*first time*/
            list[index]->length = 1;
            list[index]->head =list[index]->tail =new;
        }
        else {

```

```

        list[index]->length++;
        (list[index]->tail)->next=new;
        list[index]->tail=new;
    }
    strcpy(buffer, "    ");
    leng = 0;
    ctr++;
    p = 0;
    v = 0;
    } /* end if PUNCT(c) */
    if ((PUNCT(c)) && (p ==1))
    {
        buffer[0] = c;
        buffer[1] = '\\0';
        new = newnode();
        new->val=ctr;
        n e w - > w o r d = ( c h a r
*)malloc(sizeof(char)*(strlen(buffer)+1));
        strcpy(new->word,buffer);
        if(list[index]->length==0) { /*first time*/
            list[index]->length = 1;
            list[index]->head = list[index]->tail = new;
        }
        else {
            list[index]->length++;
            (list[index]->tail)->next=new;
            list[index]->tail=new;
        }
        strcpy(buffer,"    ");
        leng = 0;
        ctr++;
        c = getc(fid);
    } /* end second if PUNCT(c) */
} /* end for */
fclose(fid);

/* now done with the dictionary reading */
/* start to read the text to be compressed */

fid=fopen(argv[2],"r");
fhoff = fopen("hoffman","w");
fword = fopen("wordout","w");
v = 0;
ctr = ctr - 1;
strcpy(buffer, "    ");
leng = 0;
for(;;) {
    c = getc(fid);
    if (c == EOF) break;

    if (v == 0)

```

```

        {
            if (65< (int) c && (int) c <90) index = 28;
            else {
                if((int) c <97 || (int) c >122) index=27;
                else index = (int) c - 97;
            }
        }

        if((060<=c && c<=071)|| (0101<=c && c<=0132)|| (0141<=c
&& c<=0172))
        {
            buffer[leng++] = c;
            v = 1;
        }
        if ((PUNCT(c)) && (buffer[0] != ' '))
        {
            buffer[leng] = '\0';

            /* start searching */

            ptr=list[index]->head;
            FOUND=FALSE;
            while(ptr!=NULL && !FOUND) {
                if(!strcmp(ptr->word,buffer)) { FOUND=TRUE;
                    fprintf(fhoff,"%d ",ptr->val);
                    break;
                }
                else {ptr = ptr->next;}
            }

            /* if not FOUND */

            if(!FOUND) {/* add to the output */
                fprintf(fhoff,"%d ",++ctr);
                fprintf(fword,"%s ", buffer);
                new = newnode();
                new->val = ctr;
                new->word = (char
*)malloc(sizeof(char)*(strlen(buffer)+1));
                strcpy(new->word, buffer);
                if(list[index]->length == 0){
                    list[index]->length = 1;
                    list[index]->head = list[index]->tail =
new;
                }
                else {
                    list[index]->length++;
                    (list[index]->tail)->next = new;
                    list[index]->tail = new;
                }
            }
        }
    }

```

```

strcpy(buffer, "      ");
leng = 0;
v = 0;
index = 27;
}
if (PUNCT(c))
{
    buffer[0] = c;
    buffer[1] = '\\0';

    /* start searching */

    ptr=list[index]->head;
    FOUND=FALSE;
    while(ptr!=NULL && !FOUND) {
        if(!strcmp(ptr->word,buffer)) { FOUND=TRUE;
            fprintf(fhoff,"%d ",ptr->val);
            break;
        }
        else {ptr = ptr->next;}
    }

    /* if not FOUND */

    if(!FOUND) {/* add to the output */
        fprintf(fhoff,"%d ",++ctr);
        fprintf(fword,"%s ", buffer);
        new = newnode();
        new->val = ctr;
        new->word = (char *)malloc(sizeof(char)*(strlen(buffer)+1));
        strcpy(new->word, buffer);
        if(list[index]->length == 0){
            list[index]->length = 1;
            list[index]->head = list[index]->tail =
new;
        }
        else {
            list[index]->length++;
            (list[index]->tail)->next = new;
            list[index]->tail = new;
        }
    }
    strcpy(buffer, "      ");
    leng = 0;
}
}
fprintf(fword,"^A");
fclose(fword);
fclose(fhoff);

```

```

hoff_com();
connect();
printf("\n\n Compression done!");
printf("\n The compressed file called OUTPUT\n");

}

struct node *newnode()
{
struct node *tmp;

if(!(tmp=(struct node *) malloc(sizeof(struct node))))
    {printf("out of the storage\n"); exit(1);}
tmp->next=NULL;
return(tmp);
}

void      hoff_com()
{
    char          c;
    static int     counter, tempa;
    int            k, s, totbit, siz ;
    int            b=0, a0=0, a1=0, a2=0, a3=0, a4=0, a5=0,
a6=0, a7=0, a8=0, a9=0;
    float          pb=0.0, p0=0.0, p1=0.0, p2=0.0, p3=0.0,
p4=0.0;
    float          p5=0.0, p6=0.0, p7=0.0, p8=0.0, p9= 0.0;

    fred = fopen("hoffman","r+");
    for (; ;)
    {
        c = getc(fred);
        if (c == EOF) break;
        if ( c == ' ' ) b++;
        if ( c == '0' ) a0++;
        if ( c == '1' ) a1++;
        if ( c == '2' ) a2++;
        if ( c == '3' ) a3++;
        if ( c == '4' ) a4++;
        if ( c == '5' ) a5++;
        if ( c == '6' ) a6++;
        if ( c == '7' ) a7++;
        if ( c == '8' ) a8++;
        if ( c == '9' ) a9++;
    }
    fclose(fred);
    s = (b+a0+a1+a2+a3+a4+a5+a6+a7+a8+a9);
    totbit = (b*2 + a0*6 + a1*2 + a2*3 + a3*4 + a4*4 + a5*4
+a6*4 + a7*4
+a8*5 + a9*6);
    printf ("Total bits: %d\n",totbit);
}

```

```

pb = (float)(b)/(float)(s)*100;
p0 = (float)(a0)/(float)(s)*100;
p1 = (float)(a1)/(float)(s)*100;
p2 = (float)(a2)/(float)(s)*100;
p3 = (float)(a3)/(float)(s)*100;
p4 = (float)(a4)/(float)(s)*100;
p5 = (float)(a5)/(float)(s)*100;
p6 = (float)(a6)/(float)(s)*100;
p7 = (float)(a7)/(float)(s)*100;
p8 = (float)(a8)/(float)(s)*100;
p9 = (float)(a9)/(float)(s)*100;

siz = totbit/8 + 1 ;
printf("Siz %d\n",siz);
lala = (char *) malloc(siz * sizeof(char));
fred = fopen("hoffman","r+");
tempa = 0x0000;
counter = 0;
fend = fopen("teliko","w+");
for (; ;)
{
    c = getc(fred);
    if (c == EOF) break;
    bit_com(c);
}
fclose(fend);
fclose(fred);
}

void    bit_com(number)
char    number;
{
    static int    tempa, counter;
    int          mask = 0x0000, temp2 = 0x0000;
    int          cormask, bhof, m;
    int          acounter;

    switch (number)
    {
        case ' ':
            bhof = 0x0000;
            acounter = 14;
            bhof <=< acounter;
            bhof >>= counter;
            tempa = tempa | bhof;
            counter = counter + 2;
            break;

        case '1':
            bhof = 0x0002;
            acounter = 14;

```

```

        bhof <=& acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 2;
        break;

    case '8':
        bhof = 0x000b;
        acounter = 11;
        bhof <=& acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 5;
        break;

    case '2':
        bhof = 0x0007;
        acounter = 13;
        bhof <=& acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 3;
        break;

    case '7':
        bhof = 0x000d;
        acounter = 12;
        bhof <=& acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 4;
        break;

    case '3':
        bhof = 0x0004;
        acounter = 12;
        bhof <=& acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 4;
        break;

    case '6':
        bhof = 0x000c;
        acounter = 12;
        bhof <=& acounter;
        bhof >>= counter;
        tempa = tempa | bhof;
        counter = counter + 4;
        break;

```

```

        case '4':
            bhof = 0x0006;
            acounter = 12;
            bhof <=& acounter;
            bhof >=& counter;
            tempa = tempa | bhof;
            counter = counter + 4;
            break;

        case '9':
            bhof = 0x0014;
            acounter = 10;
            bhof <=& acounter;
            bhof >=& counter;
            tempa = tempa | bhof;
            counter = counter + 6;
            break;

        case '5':
            bhof = 0x0007;
            acounter = 12;
            bhof <=& acounter;
            bhof >=& counter;
            tempa = tempa | bhof;
            counter = counter + 4;
            break;

        case '0':
            bhof = 0x0015;
            acounter = 10;
            bhof <=& acounter;
            bhof >=& counter;
            tempa = tempa | bhof;
            counter = counter + 6;
            break;
    }
    if ( counter >= 8 )
    {
        temp2 = tempa & 0xff00;
        temp2 >=& 8;
        if (temp2 == 0x1a) temp2 = 0x1;
        if (temp2 == 0x00ff) temp2 = 0x81;
        *lala = temp2;
        fprintf(fend,"%c",*lala);
        lala++;
        tempa <=& 8;
        counter = counter - 8;
    }
}

```



```

void    connect()
{
    char        c, d;

    ffin = fopen("output","w+");
    fwor = fopen("wordout","r+");
    fnum = fopen("teliko","r+");

    for (;;)
    {
        c = getc(fwor);
        if ( c == EOF) break;
        putc(c,ffin);
    }
    for (;;)
    {
        d = getc(fnum);
        if ( d == EOF) break;
        putc(d,ffin);
    }
    fclose(ffin);
    fclose(fwor);
    fclose(fnum);
}

```

REFERENCES

1. Bell C. T., Better OPM/L Text Compression, IEEE Transactions on Communication December 1986.
2. Bentley L. J., Sleator D. D., Tarjan E. R. and Wei K. V., A Locally Adaptive Data Compression Scheme, Communication ACM April 1986.
3. Cleary G. J. and Witten H. I., Data Compression Using Adaptive Coding and Partial String Matching, IEEE Transactions on Communication, April 1986.
4. Hamming W. R., Coding and Information Theory, Prentice Hall 1986.
5. Kucera H., Francis W. N., Computational Analysis of Present-Day American English, Brown University Press, Providence, Rhode Island 1967.
6. Langdon Jr G. G., An Introduction to Arithmetic Coding, IBM Res. Develop, Vol 28 No 2, March 1984.
7. Storer A. J., Data Compression methods and theory, Computer Science Press 1988.
8. STAC ELECTRONICS INC. Carlsbad CALIFORNIA 92008.
9. Tanenbaum A. S., Computer Networks, Prentice Hall, 1988.
10. UNIX
11. Welch A. T., A Technique for High-Performance Data Compression, IEEE June 1984.
12. Witten H. I., Neal M. R. and Cleary G. J., Arithmetic Coding For Data Compression, Communication ACM June 1987.
13. Ziv I. and Lempel A., Compression of Individual Sequences via Variable-Rate Coding, IEEE Transaction on Information Theory September 1978.